UCRL-ID-137121

Users Manual for the UEDGE Edge-Plasma Transport Code

T.D. Rognlien, M.E. Rensink, G.R. Smith

January 10, 2000

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Users Manual for the UEDGE Edge-Plasma Transport Code

T.D. Rognlien, M.E. Rensink, and G.R. Smith

Lawrence Livermore National Laboratory

Livermore, CA 94551

Operational details are given for the two-dimensional UEDGE edge-plasma transport code. The model applies to nearly fully-ionized plasmas in a strong magnetic field. Equations are solved for the plasma density, velocity along the magnetic field, electron temperature, ion temperature, and electrostatic potential. In addition, fluid models of neutrals species are included or the option to couple to a Monte Carlo code description of the neutrals. Multi-species ion mixtures can be simulated. The physical equations are discretized by a finite-difference procedure, and the resulting system of algebraic equations are solved by fully-implicit techniques. The code can be used to follow time-dependent solutions or to find steady-state solutions by direct iteration.

Contents

1. INTRODUCTION

This report gives the operational details of how to use the UEDGE code. A brief description of the equations solved is included in the Appendix; for more details, the reader can refer to the reference list at the end of this report which include more information on the models and examples of the results obtained with the code.

UEDGE is a two-dimensional (2D) fluid transport code for collisional edge plasmas. Its primary use has been for tokamak edge plasmas in Magnetic Fusion Energy devices, mainly tokamaks, although linear devices and spheromaks have also been modeled. UEDGE typically generates a curvilinear mesh based on the poloidal flux surfaces from an MHD equilibrium code such as EFIT or TEQ, but there are options for Cartesian meshes and cylindrical meshes as well.

The basic physics equations are taken from Braginskii [1], with the addition of *ad hoc* anomalous or turbulence-enhanced transport coefficients for the direction across the magnetic field; transport along the magnetic field is taken as classical with flux limits. A discussion of the rationale for this procedure is given in Ref. [2]. Also, an arbitrary number of ion species can be included (limited by computer memory and speed) which goes beyond Braginskii's model. Line-radiation loss from excitation, ionization, and recombination is incorporated into the electron energy equation. Neutral gas is described by fluid models or by coupling to a Monte Carlo code.

At its inception in 1992, UEDGE used a set of basic physics equations and finite-differencing similar to that in the original B2 transport code [3,4]. However, UEDGE uses a fully implicit procedure known as a modified Newton iteration to solve all of the equations simultaneously rather the the semi-implicit SIMPLE algorithm used in B2. Overviews of the fully-implicit method used in UEDGE are given in Refs. [5,6]. In addition, UEDGE, now includes a detailed fluid neutral model with a parallel momentum equation, calculation of the electrostatic potential with $\mathbf{E}\times\mathbf{B}$ and ∇B drift effects, and a nonorthogonal mesh to conform to shaped divertor surfaces.

References 7–25 give many details of the models used in UEDGE and some applications for edge-plasmas in fusion devices. The list is not intended to be exhaustive, but to serve as beginning bibliography for those seeking more detail.

2. Source Code, Variable Descriptor Files, and Building an Executable

2.1. Source code

The source code is maintained in a CVS archive on the MFE Program network at LLNL. Precompilation processing can be done on any workstation that can access to this network, and others can obtain the necessary files by contacting the authors. Compilation and loading is done on the machine where UEDGE is to be run. Details of this procedure are given below in Sec. 2.3.

UEDGE is divided into ten BASIS packages with different functions. For example, the finite-differenced physics equations are contained in the package **bbb**, the grid generation routines are in packages **flx** and **grd**, and the (**pfb**) package (provided by BASIS) allows reading and writing data in a portable format PFB save file. An alphabetical list and summary of all of the packages (each having its own directory) and several important auxiliary directories is as follows:

aph	# calculates atomic cross-sections for hydrogen
api	# calculates atomic cross-sections for impurities
bbb	# sets up the finite-difference physics equations and routines
	# needed by the linear algebra solvers - the heart of UEDGE
com	# contains routines and variables needed by various packages
dce	# distributed computing package - not generally needed
doc	# documentation including UEDGE manual, uedge.man
flx	# calculates the magnetic flux surfaces for mesh
grd	# calculates second mesh coordinate and constructs the mesh
in	# various diagnostic routines (not a package)
scripts	# BASIS scripts for finding files (not a package)
svr	# linear algebra and temporal integration routines
test	# a few test cases (not a package)
wdf	# calculates data needed by the DEGAS2 neutral M.C. code

2.2. Variables

All of the variables within UEDGE, together with a one-line description of their meaning, are listed in files called the variable descriptor files which are used in conjunction with the BASIS code-development system. Generally, there is one such file for each package listed above, with the name of the file being package_name.v in the directory package_name. For example, variables for the physics equations and routines for the numerical Jacobian in UEDGE are included in the variable descriptor file called bbb.v in directory bbb. The variables are combined into groups to make the process of identification easier. Also, if you know the name of the variable while running UEDGE, at the UEDGE> prompt, you may type list xyz, and you will receive the information about that variable included in the variable descriptor file. Also, if you know the name of the group, you can type list groupname, and receive a listing of that group from the variable descriptor file.

The primary plasma fluid variables used in the code are as follows:

ni(0:nx+1,0:ny+1,nfld)	Ion dens. [m**(-3)], nfld=species index (default=1)
up(0:nx+1,0:ny+1,nfld)	Parallel ion flow velocity [m/s]
te(0:nx+1,0:ny+1)	Electron temperature [Joules]
ti(0:nx+1,0:ny+1)	Ion temperature [Joules]
ng(0:nx+1,0:ny+1,ngsp)	Neutral gas density [m**(-3)], ngsp=species index
phi(0:nx+1,0:ny+1)	Electrostatic potential [Volts]

Note that SI units are used throughout UEDGE. The first two indices for each variable correspond to mesh indices (ix,iy), where ix is the poloidal index beginning at the inner divertor plate for a tokamak, and iy is the radial-like index beginning at the core boundary or the private-flux-region wall.

The primary plasma fluid variables are used to evaluate the spatial derivatives and source terms in the PDE's. However, the variables that are passed to the ODE and Newton solvers are somewhat different and normalized. These are as follows:

For density:	ni(i)/n0(i), where n0(i) is an input constant
For velocity:	ni(i)*up(i)/[n0(i)*cs], where cs is a constant ion-acoustic speed
For Te:	ne*te/(n0(1)*tnorm), where tnorm is a constant, ne = elec. dens.

For Ti:	ne*ti/(n0(1)*tnorm), where tnorm is a constant, ne = elec. dens.
For ng:	ng(i)/n0g(i)
For phi:	ev*phi/tnorm, where ev=1.6e-19 is the electron charge

The conversion from plasma variables to ODE variables is done in subroutine convrs; the reverse conversion is done by subroutine convert.

The ODE variables are stored in a 1-D vector call yl, starting at ix=0, iy=0. The variables at that point are stored in the order listed above, then the poloidal index, ix, is incremented until the ix=nx+1 boundary is reach, then iy is incremented by unity, and the process repeated. There are index arrays that allow the user to determine the index ieq of yl(ieq) that corresponds to a variable at a given (ix,iy) location on the grid; these arrays are defined as follows:

idxn(ix,iy,ifld):	yl ieq index for ni(i)/n0(i)
idxu(ix,iy,ifld):	yl ieq index for ni(i)*up(i)/[n0(i)*cs]
idxte(ix,iy):	yl ieq index for ne*te/[n0(1)*tnorm]
idxti(ix,iy):	yl ieq index for ne*ti/[n0(1)*tnorm]
idxg(ix,iy,igsp):	yl ieq index for ng(i)/n0g(i)
idxphi(ix,iy):	yl ieq index for ev*phi/tnorm

There are also two arrays that the give the ix and iy indices for a given yl index ieq:

igyl(ieq,1):	ix poloidal index for ODE variable ieq
igyl(ieq,2):	iy radial index for ODE variable ieq

2.3. Compiling and loading a new executable

The procedure for compiling and building the UEDGE code is outlined below:

Begin in your home directory, and un-tar the files containing UEDGE by the command

```
tar xvf uedgeXXXXXX.tar
```

where XXXXXX will be the date the tar file was generated; for example, uedge092399.tar.

Set the following environmental variables; note that BASIS_ROOT and NCAR G_ROOT will depend on where the system administrator has stored BASIS and NCAR graphics 0n your computer system

```
setenv UEDGE_SCRIPTS ~/uedge/scripts
setenv BASIS_ROOT /usr/local/nbasis
setenv NCARG_ROOT /usr/local
setenv OPT '-native -03'
set path = ($BASIS_ROOT/bin $path)
```

Next, go to the ~/uedge directory and issue the following to generate the required make files

```
mmm -ezn -rl
```

where ezn is the graphics package and rl is the readline library that allows line editing and line recall. This command generates the makefiles needed to compile and load UEDGE.

Finally, type the following line to compile and load the executable code xuedge

```
gmake all xuedge
```

The excecutable xuedge will appear in your uedge/SOL directory if you are on a SUN system.

3. Basic Mechanics of Code Execution

3.1. Reading input and execution

The executable for UEDGE is typically called xuedge, or sometimes just uedge on NERSC machines. The build and loading procedure is described in Sec. 2.3, which results in the executable being found in ~developer/uedge/SOL, where developer is the name of the person who created the executable and SOL refers to the SUN Solaris system. On other workstations, SOL will be replaced by something like HP700 for HPs or AXP for DECs, etc.

Before running UEDGE or xuedge, you need to set two environmental variables if you have not already done so for the compilation and loading of UEDGE described in Sec. 2.3. If you are running on the LLNL MFE SUN solaris system, you can just set two environmental variables: UEDGE_SCRIPTS to /home/rognlien/uedge/scripts and UEDGE to /home/rognlien/uedge. Then you need not worry about the aphdir and uedge_path files. If you are on another system, or want to read your own atomic physics and script files, place the contents of uedge/in/aph and uedge/in/api in some directory, and then set the environmental variable UEDGE to point to them. Likewise, copy and then change what is in uedge/scripts and set up the environmental variable UEDGE_SCRIPTS to point to that directory.

The aphdir file tell UEDGE where to locate the hydrogen atomic rate files, and on the LLNL MFE SUN solaris system, it should contain the line

```
aphdir = "/mfe/theory/Uedge/Ver_XX/uedge/in/aph"
```

where the XX is the most current version number shown in the Uedge directory. The second file, uedge_path, tells UEDGE where to find various diagnostic files. Again, on the LLNL MFE SUN solaris system, it should contain the line

```
call pathadd("/mfe/theory/Uedge/Ver_XX/uedge/in")
```

One can add other lines to tell UEDGE to search other paths (directories) in response to a command issued from the parser.

To run a case, simply type xuedge (or uedge if you are using the public version on the NERSC system). The executable will then automatically read a default input file called .basis, if it exists in the directory (generally not used, however). After UEDGE has read the .basis file, it comes back with the prompt UEDGE>, at which point you need to type a command. Typically, you will read a second file which contains the input settings for various control variables and switches; we usually use the convention that such input files begin with the letters rd (which stands for "read"), but this is not necessary. Thus, you will type something like read rddata2. You may also start the executable and read the input file on a single line with the command xuedge read rddata2. Some example files can be found in the location noted below in Sample Problems section. You can also change any variable at the prompt by typing, say, runtim=1.e-10, to modify runtim.

Once all the variables have been set, you execute the code by typing exmain. What this does is tell the BASIS system to execute the subroutine named exmain, which calls all the appropriate subroutines to execute a full run. It is also possible under BASIS to call other subroutines independently, but this is usually only done for debugging purposes. Thus, a typical session with UEDGE will look as follows:

```
xuedge
UEDGE> read rddata2
UEDGE> exmain
UEDGE>
```

The last prompt means the code has successfully completed the run and is ready for more input. If you want to stop, just type end. You can also display the results without exiting from the BASIS session.

3.2. Sample problems

There are a set of sample problems that can be run. For example, on the LLNL MFE SUN system, one can go to the directory ~rognlien/uedge/Applic/Examples. It is best to check with one of the current users to learn more details of these examples.

3.3. Displaying results

You may want to list any variable or array to see what the solution looks like. For example, the 2-D electron temperature is stored in the array te(0:nx+1,0:ny+1) in units of Joules. You may specify the number of places to be displayed after the decimal point by typing fuzz=2, for example. Thus, the following sequence will print out the electron temperature array in electron volts:

```
fuzz=2
te/ev
```

where ev=1.6e-19 Joules/eV is a variable in the code for converting from Joules to electron volts. If you want to list some other variable to 4 decimal places you must first type fuzz=4, which will hold until another fuzz= statement is typed.

You can also do plots of arrays to look at the present solution. To plot the function y(x), type plot y,x. To do a contour plot of z(x,y), type plotz z,x,y.

All of the commands useable at the UEDGE> prompt are in the BASIS command language, which is extensive and powerful yet easy to learn. The BASIS manual can be read and searched with a web browser (see the URL http://xfiles.llnl.gov/basis/). The plotting package EZN is also part of BASIS.

3.4. *Instantaneous output and run termination*

While running a time-dependent problem with any solver (except lsode, which is not a currently supported option), you may obtain information on the present status of the solution by typing s or status and a return. You will receive lines of output giving the total number of function and preconditioner evaluations, nfe and npe, respectively, and the yl index ieq (imxtstep) of the equation that is restricting the time step. A second index, imxnewt, gives the ieq of the equation that is requiring the Jacobian to be reevaluated, if that is limiting the time step. In addition, it gives the total simulation time and the present time step, dt. A third line gives detail data on the error estimates from the ODE solver: bigts is for the time integration and dsm is for the Jacobian.

If you want to abort the present simulation and return to the UEDGE> prompt, type ctrl-c and a return. At this point, BASIS with give the prompt DEBUG>; you now may query UEDGE for variable values, etc., and then type cont if you want to continue the calculation. If you type abort, to the DEBUG> prompt, you will return to the UEDGE> prompt. You can do this for DEBUG> during either a time-dependent solution or a Newton iteration. If you then restart after changing some parameters, the code initializes itself as though the aborted run had not taken place.

3.5. *Restarting from present solution*

If you are still in an active BASIS session, you can set restart=1 to use the present solution as initial conditions for the next execution. You can also double the grid in both x and y directions by being sure that newgeo=1, restart=1 and by doubling nxleg, nxcore, nycore, nysol, and nxomit. The present solution is then interpolated to the finer grid as the initial conditions for the new run. This is conveniently done by reading a file called double with BASIS (i.e., type read double) that has the necessary parameter adjustments in it. You can double the mesh separately in the radial or poloidal direction, or incrementally; see Sec. 3.7 for more details.

3.6. *Restarting from a* BASIS *portable PFB save file*

BASIS has a very useful facility to save any variables you wish to a portable file that can be read in a subsequent session. To do this, type create sfile, where sfile is some name you choose for the data file. Then you can save any variables you want by typing write x,y,te. This command can be repeated; to stop the writing to this file, type close. The minimum data set you must save in a portable file to make a restart is the set of plasma variables; you should also save any special variable settings. An example for the minimum saving procedure is as follows:

```
create stest1
write nis,ups,tes,tis,ngs,phis
close
```

UEDGE uses the convention that the plasma variables used for restarting all have the letter "s" appended to them.

To use the portable file, you must be sure that you have the same grid size as that of the saved data. You then execute UEDGE as for a normal run by reading input files (but don't type exmain yet). Then give the following commands:

```
allocate
restore stest1
restart=1
```

Here, allocate generates the appropriate arrays through dynamic allocation. At this point, one may type exmain to begin the run with the values saved previously in stest1. It is convenient to save the final state of converged runs in case you want to restart from them at some later date. Also, the portable files can be moved (using the binary mode of ftp) to computers using different numerical representations and used to continue runs.

Before version 10.0 of BASIS, a capability was present to create savefiles using commands save to, save, and save off; users of more recent vesions do not need to concern themselves with this difference. The portable file capability (PFB or PDB) should be used now to create new files, but the user may want to use a version of UEDGE created with version 9.11 of BASIS to convert from the savefile format to the portable format. Use the readb command to read a savefile just as you would use restore to read a portable file. Then use create, write, and close to save the data in a portable file.

3.7. Interpolating the solution to a new mesh and restarting

UEDGE has three different linear interpolating options; these are controlled by the switches isnintp and isgindx. If isnintp=0, isgindx is immaterial, and the "old" interpolator is used that only allows the users to double the mesh in each direction. That is, if you are starting from a solution with a certain set of grid indices nxleg(1,1), nxleg(1,2), nxcore(1,1), nxcore(1,2), nysol(1), and nycore(1), you must double all of these. This interpolation is rather crude in that it assumes the mesh is uniform in each direction, but it works surprisingly well. However, doubling the mesh in each direction is sometimes too large a change for the Newton method to converge on the finer mesh. The next two methods allow an arbitrary change in the number of mesh points.

If isnintp=1 the user may increase or decrease the mesh by any amount in either direction. For this case, two options are available: isgindx=1 and 0. For isgindx=1 (default and recommended setting), the interpolation occurs in index space as though the mesh is uniform. This case is thus similar to the isnintp=0 option discussed above. However, there is a difference (other than the arbitrary mesh change) in that here values are not interpolated across the separatrix or across the radial cut through the x-point, but are rather extrapolated at these locations. The reason for doing this across the separatrix is that linear interpolation often does a poor job because of the abrupt change in variables there; it is done across the radial cut only for simplicity of the algorithm. Treating the region above and below the separatrix independently (and on either side of the cut) allows one to add extra mesh points in either region without perturbing the other.

For isnintp=1 and isgindx=0, the code does a linear interpolation using the actual (normalized) mesh which is not uniform or rectangular. This scheme does not seem to outperform the simpler index-based algorithm, and sometimes has trouble finding the appropriate mesh points for performing the interpolation. The message ***** grdinty cannot find straddling grid... is printed if the algorithm fails to find the appropriate mesh points. In that case, switch to isgindx=1.

It is possible to interpolate from a save-file solution, even if the code does not converge to this case first. One needs to generate the old mesh and then type call gridseq from the UEDGE prompt (the BASIS parser). Alternatively, use a very loose tolerance for svrpkg="nksol", say ftol=1.e10, so the code will think it has converged after one iteration. The mesh may then be changed without any extra call to gridseq.

4. Grid Generation

4.1. *Mesh generation using MHD equilibria*

The grid generated in UEDGE uses routines that are an extension and modification of a grid generator developed by M. Petravic at PPPL [26]. Data on the location of poloidal magnetic flux surfaces are generated by one of various MHD equilibrium codes (*e.g.*, TEQ, EFIT), and then read into UEDGE via two files. These files must be named aeqdsk and neqdsk, and the switches mhdgeo=1 and gengrid=1 must be set. If gengrid=0, UEDGE reads in a file with the grid information already in it called gridue. The gridue file can be generated by a previous UEDGE run. For large problems, precomputing the gridue file and using gengrid=0 can save considerable storage. One can generate the grid in UEDGE without running a complete problem; just type

```
call flxrun
call grdrun
```

and the file gridue with be generated. This call sequence is done automatically if you execute a full problem by typing exmain.

There are a number of options available for the grid generation; we mention a few here. For single-null configurations, set geometry="snull"; for double null we use the lower half and set geometry="dnbot". It is also possible to simulate the outer half of a single null or the lower, outer quadrant of a double null where reflection boundary conditions are used at the left boundary and no flux is allowed through the outer (ix=ixpt2) cut at the x-point. To do this latter type of geometry, set these variables:

```
nxomit        # Number of poloidal grid points to omit before setting
              # reflection boundary condition. Files double, etc.
              # automatically change nxomit to correct value
isfixlb = 2   # Sets reflection bc at ix = nxomit and no-flux bc at
```

```
# ix = ixpt2 (outer x-point cut)
```

The radial distribution of the mesh is controlled by the following input parameters. The poloidal flux, psi, is normalized to be unity on the separatrix and the radial boundaries of the mesh are specified by:

psi0min1	:flux at the inner core boundary, 0.98 is typical
psi0min2	:flux at the private-flux region boundary, 0.98 is typical
psi0sep	:flux very near the separatrix, use 1.00005
psi0max	:flux at the outer wall boundary, 1.07 is typical

The number of radial cells in various regions is:

nycore(1)	:number of radial cells in core (and private-flux) region
nysol(1)	:number of radial cells in scrape-off layer

The input variable alfcy provides some additional control over the radial distribution of mesh points. One can cause the flux surfaces to cluster near the separatrix by making the variable alfcy somewhat larger than unity (2-3 is typical); if alfcy = 0, the radial mesh in the SOL is almost uniform except near the x-point.

For double-null configurations the radial distribution of flux surfaces can be different for the inboard and outboard legs of the plasma. In this case psi0max and alfcy refer to the outboard leg and one must supply additional input for

psi0max_inner	:flux at 'outer' wall boundary on inboard leg
alfcy_inner	:cluster factor for flux surfaces on inboard leg

The poloidal distribution of the mesh is controlled by the following input parameters. The mesh ends at the divertor plates which are defined by the separatrix strike points included within in the input file aeqdsk. The mesh is divided into various regions, and one can specify the number of cells in each as follows:

nxleg(1,1)	:number of poloidal cells in inboard leg between divertor plate and x-point
nxcore(1,1)	:number of poloidal cells in inboard region of core

	:between x-point and top (or midplane for double-null)
nxcore(1,2)	:number of poloidal cells in outboard region of core
	:between x-point and top (or midplane for double-null)
nxleg(1,2)	:number of poloidal cells in outboard leg between divertor plate and x-point

Additional control over the distribution of the mesh along the separatrix in the poloidal or x-direction is provided by a function x(t) where t is the indexing parameter that labels the cells; the grid points are spaced uniform in t for a given region (leg, or core), and the actual spacing in x is determined by x(t). There are a number of options for x(t) controlled by the switch kxmesh:

kxmesh=0	:manual definition of seed points
kxmesh=1	:use linear*rational form for x(t) in divertor
kxmesh=2	:use linear*exponential form for x(t) in divertor
kxmesh=3	:use spline form for x(t) everywhere
kxmesh=4	:use exponential+spline form for x(t) in divertor

For kxmesh=0, one must fill the arrays seedxp and seedxpxl. These arrays give the location on the separatrix of the mesh points in percentage of the distance from the x-point to plate or x-point to top of machine, etc; thus, all values must lie between 0-100 and be monotonic. A standard procedure for setting the arrays seedxp and seedxpxl is to generate an approximate mesh with a kxmesh.ne.0 option, and then read the files rdgen.seedxp in directory uedge/in. This fills the arrays with the current mesh values, and one can then edit these arrays by inserting or moving points. The resulting arrays should be saved into a PFB file which can then be read in using the restore command for generating the desired mesh with kxmesh=0.

For kxmesh=1 or kxmesh=2 the poloidal spacing of the cells between the x-points is nearly constant unless one makes the variable slpxt larger than unity (1.2 to 1.3 is typical); having slpxt $>$ 1 causes the cells to cluster near the x-point on both sides, and thus often match more smoothly with the cells in the divertor leg regions.

For kxmesh=4 the user specifies sub-regions in front of each divertor plate: the variables nx-gas(1:2) specify the number of cells in the region, the variables dxgas(1:2) specify the size of the first cell at the divertor plate, and the variables alfx(1:2) specify the exponential factor for the cell-to-cell variation in the region. The relation between dxgas and the total length of the exponential region, L, is given by L = dxgas*[exp(alfx*nxgas)-1]/[exp(alfx)-1].

4.2. Non-orthogonal grids

Non-orthogonal grids can be generated by setting the switch ismmon:

ismmon=0	:strictly orthogonal mesh (default)
ismmon=1	:poloidal mesh is compressed/expanded w.r.t. orthogonal
ismmon=2	:poloidal mesh varies smoothly on each flux surface
ismmon=3	:combination of ismmon=0,1,2

This switch affects the distribution of meshpoints on each flux surface, but does not alter the the number or spacing of the flux surfaces.

For ismmon=1 the mesh is generated by deforming a previously generated orthogonal grid along flux surfaces that intersect the divertor plates. The original mesh is uniformly compressed or expanded in the poloidal direction until the end of the mesh just coincides with the divertor plate. The compression or expansion occurs along each flux surface between some upstream reference surface (such as the midplane) and the divertor plate. A smoothing procedure may subsequently be applied to remove abrupt distortions in the mesh.

Options under the ismmon=1 setting are:

istream	:definition of the upstream reference surface
	:=0 (default) –> midplane in SOL, cut in p.f. region
	:=1 user-defined
iplate	:definition of the divertor plate surface
	:=0 (default) –> orthogonal plate
	:=1 user-defined
nsmooth	:number of smoothing passes applied to each surface
	:=0 –> no additional mesh modification
	:=2 (default) recommended

The user defines the divertor plates via the arrays

```
rplate1(1:nplate1)      zplate1(1:nplate1)
```

for the inboard leg of the divertor, and

```
rplate2(1:nplate2)      zplate2(1:nplate2)
```

for the outboard leg of the divertor. Usually, one would read this information from a text file prepared specifically for the device being modelled. Examples of such files for DIII-D, CMOD, TPX, and ITER are available from the authors. NOTE: For complicated divertor geometries it may be necessary to simplify the divertor plate definition to avoid intersecting any flux surface more than once. The current version assumes there is only one intersection.

The user defines the fixed upstream reference surface via the arrays

```
rupstream1(1:nupstream1)      zupstream1(1:nupstream1)
```

for the inboard half of the mesh, and

```
rupstream2(1:nupstream2)      zupstream2(1:nupstream2)
```

for the outboard half of the mesh. Usually, one would read this information from a previously prepared text file. For example, on open SOL flux surfaces one might choose to modify the mesh only downstream from the midplane and on private flux surfaces only downstream from the "cut" under the x-point. The file that does this is upstream.mpc available from the authors. The mesh in the core region would not be modified.

EXAMPLE: In addition to parameters for an orthogonal mesh, set the following:

```
ismmon=1          # switch on mesh modification
istream=0         # use default upstream reference surface
iplate=1          # flag indicates user will supply plate definition
read plate._device# define divertor plate surfaces
nsmooth=2         # use default smoothing of "radial" surfaces
```

For ismmon=2 the normalized poloidal distribution of mesh points is the same on each flux surface. The distribution on the separatrix flux surface is defined according to the input parameter 'kxmesh' described earlier in this writeup. This poloidal distribution is then normalized in terms of the total poloidal connection length from the divertor plate surface to the top of the mesh (for open SOL flux surfaces) or the cut under the x-point (for private flux surfaces). Then, on each flux surface the poloidal distribution of mesh points is obtained by scaling the normalized separatrix distribution with the poloidal connection length for that surface. The resultant mesh is non-orthogonal even for orthogonal divertor plates.

Options under the ismmon=2 setting are:

istream	:definition of the upstream reference surface
	:=0 (default) –> top of the mesh in SOL, cut in private flux region
	:=1 user-defined
iplate	:definition of the divertor plate surface
	:=0 (default) –> orthogonal plate
	:=1 user-defined
nsmooth	:number of smoothing passes applied to each "radial" surface
	:=0 –> no additional mesh modification
	:=2 (default) recommended

EXAMPLE:

In addition to parameters for an orthogonal mesh, set the following:

```
ismmon=1             # switch on mesh modification
istream=0            # use default upstream reference surface
iplate=1             # flag indicates user will supply plate definition
read plate._device_  # define divertor plate surfaces
nsmooth=2            # use default smoothing of "radial" surfaces
```

4.3. Adaptive-mesh capability

The mesh can be modified in response to the plasma state so as to obtain better resolution in spatial regions where physics variables are changing most rapidly. At present, this capability is limited to a poloidal re-distribution of mesh points along each flux surface. The number and position of the flux surfaces is not changed, i.e. the “radial” resolution is fixed. The basic idea is to poloidally refine the mesh near a “flamefront” surface between the x-point and the divertor plate(s). This process is not yet automated so the user must manually perform certain steps to change the mesh and then obtain a plasma solution on the modified mesh.

The user-callable subroutine meshff(region) modifies a reference mesh stored in arrays (cmeshx3, cmeshy3) and writes the modified mesh into the arrays (cmeshx,cmeshy). Here, region=1 for the inboard leg and region=2 for the outboard leg. The mesh is modified only between the x-point and the divertor plate(s); the core and adjacent SOL regions of the mesh are unchanged. It is the user’s responsibility to store the appropriate data in (cmeshx3,cmeshy3) before calling meshff. After meshff completes, it is necessary to call subroutine writeue which converts the (cmeshx,cmeshy) data into (rm,zm) data and writes the file gridue that is read by the plasma package when gengrid=0.

The flamefront surface is defined by the user via the arrays rff1(1:nff1) and zff1(1:nff1) where (rff1,zff1) are the (R,Z) coordinates [m] of the nff1 data points. The ’1’ here refers to the inboard divertor leg; there are corresponding variables with ’1’–>’2’ for the outboard divertor leg. Storage for the arrays rff1 and zff1 is dynamically allocated by setting nff1 and then calling gchange(“grd.Mmod”,0) from the parser.

Input data for the flamefront (FF) mesh modification includes -

isxtform	:a flag for choosing one of three possible forms for
	:the distribution of mesh points along a flux surface
iswtform	:flag for combining the original and FF meshes with constant
	:weight factor (iswtform=0) or index-dependent weight factor (iswtform=1)
cwtff	:shape factor for the iswtform=1 option
	:on combining original and FF meshes.

For the inboard leg:

nff1,rff1(),zff1()	:number of data points and (R,Z) coordinates of FF.
slpxff1	:slope reduction factor for x(ix) at FF position
	:on each flux surface.
slpxffu1	:slope reduction factor for x(ix) at position upstream of FF on each flux surface.
slpxffd1	:slope reduction factor for x(ix) at position

	:downstream of FF on each flux surface.
nxdff1	:number of cells between FF and divertor plate on each flux surface.
wtff1	:maximum weight factor for combining original and FF meshes.

For the outboard leg: replace 1 by 2 in the variable names above.

The input data controls the form of the meshpoint distribution along each flux surface by specifying various shape factors for an analytic function x(t) that gives the poloidal distance from the x-point as a function of the poloidal meshpoint index. Let (t1,x1) represent the x-point, (t2,x2) the flamefront, and (t3,x3) the divertor plate.

For isxtform=1 we use a piece-wise functional form; on $t1 < t < t2$ use the rational function:

x(t) = x1 + (x2-x1)*(t-t1)/((t2-t1)+alpha*(t2-t)

and on $t2 < t < t3$ use the rational function:

x(t) = x2 + (x3-x2)*(t-t2)/((t3-t2)+beta*(t3-t))

where alpha and beta are chosen to give a specified slope at t2. The slope is expressed as the product of the average slope and a slope reduction factor slpxff, x'(t2) = slpxff * (x3-x1) / (t3-t1) where a '1' or '2' should be appended to 'slpxff' for the appropriate divertor leg.

The isxtform=2 option uses a slightly more general form for x(t) which allows the user to also specify the slope at the upstream point (t1,x1) in the form x'(t1) = slpxffu * (x3-x1) / (t3-t1) where a '1' or '2' should be appended to 'slpxffu' for the appropriate divertor leg.

The isxtform=3 option uses a similar form for x(t) which allows the user to specify the slope at all three data points via the slope factors slpxff, slpxffu and slpxffd:

x'(t1) = slpxffu * (x3-x1) / (t3-t1)
x'(t2) = slpxff * (x3-x1) / (t3-t1)
x'(t3) = slpxffd * (x3-x1) / (t3-t1)

where a '1' or '2' should be appended to 'slpxffu','slpxff' and 'slpxffd' for the appropriate divertor leg.

To facilitate a gradual transition from the original mesh to a flamefront modified mesh, the two meshes are combined via a weight function, wt(t), to produce the final form of the meshpoint distribution x(t):

```
x(t) = wt(t) * xFF(t) + (1-wt(t)) * x0(t)
```

where x0(t) represents the original mesh and xFF(t) represents the flamefront mesh defined by the isxtform options above. The form of the weight factor is controlled by the flag iswtform: iswtform=0 -> constant wt(t)=wtff and iswtform=1 -> smooth increase from 0 at x-point to wtff at flamefront, where '1' or '2' should be appended to the 'wtff' for inboard or outboard regions.

The input parameter nxdff controls the number of cells downstream from the flamefront on each flux surface. At present, this number is the same for all flux surfaces. If nxdff=0, the number of downstream cells for the flamefront mesh is set equal to the number of downstream cells on the separatrix flux surface of the original mesh; otherwise, the user-specified value of nxdff sets the number of downstream cells on the flamefront mesh.

In summary, the steps necessary to use the adaptive mesh facility are:

1. define the reference mesh via the grd package arrays (cmeshx3,cmeshy3).
2. define the flamefront surface for each divertor leg.
3. set various flamefront mesh control parameters.
4. call subroutine meshff(region).
5. call writeue to convert the (cmeshx,cmeshy) data to (rm,zm).
6. set gengrid=0 (or mhdgeo=0 for cartesian configurations) and isnonog=1 before executing with exmain.

4.4. Adding poloidal cells near the x-point

The user may add extra poloidal cells near the x-point by setting the variables nxxpt and nxmod. Here nxxpt is the number of extra cells added between the x-point and the poloidal face nxmod indices away from the x-point for each of the four quadrants of a single-null divertor. This then results in a total of 4*nxxpt extra poloidal cells given by total poloidal cells =nxleg(1,1)+nxleg(1,2)+nxcore(1,1)+nxcore(1,2)+4*nxxpt

nxmod should be 1 or greater, with nxmod=2 recommended; if nxmod=2, the two cells poloidally adjacent to the x-point are recalculated including the number of extra mesh points, nxxpt. Note that these cells are not strictly orthogonal, so it is safest to use the nonorthogonal difference stencil (isnonog=1), but the error in not doing so may be small and is limited to the modified x-point region.

There is some control over the spacing of these extra cells through the variables alfxpt and alfxpt2; alfxpt controls the nonuiformity of the poloidal mesh in the modified region, and alfxpt2 controls how rapidly the poloidal face shape returns to a smooth arc as one moves away from the x-point. The practical range of alfxpt is roughly 0.25 < alfxpt < 1, where alfxpt=1 gives uniform spacing (the default) and alfxpt=0.5 gives the cell faces closer to the x-point by roughly 0.707. The range of alfxpt2 is 1 < alfxpt2 < 2, where higher values force the mesh to return to a smooth arc faster. It is best to try a few values and then look at the result by plotting the mesh with the plotmesh script.

4.5. Top-of-mesh/limiter option

By default, the spatial extent of the "inboard" and "outboard" regions of the core/SOL are delimited by a vertical line from the magnetic axis upward through the separatrix. This top-of-mesh/limiter position can be changed by setting the switch islimon=1 and defining the poloidal angle of the new top-of-mesh/limiter position, theta_lim. Theta_lim should lie with the limits -pi < theta_lim < pi and should not be too close to the x-point position. The outboard midplane position corresponds to theta_lim=0 and the default is theta_lim=pi/2. The islimon=1 option automatically turns on a procedure for checking the angular position of every data point on every flux surface, so the flx package may run slower.

4.6. Cartesian and cylindrical configurations

In addition to generating a mesh obtained from an MHD equilibrium code or model, UEDGE has the ability to simulate simpler configurations, namely, either Cartesian or cylindrical geometries. These options are controlled by the input variable mhdgeo; mhdgeo=-1 is used for a Cartesian configuration and mhdgeo=0 is used for a cylindrical configuration.

For the Cartesian case, one can set the following mesh parameters:

radx	:position outer "radial" wall, across-B-field direction
radm	:position of inner "radial" wall
rad0	:position of separatrix
alfyt	:radial nonuniformity factor; $< 0 \rightarrow$ expanding
isfixlb	:set left poloidal boundary as sym. plane
za0	:"poloidal" symmetry plane location
zaxpt	:"poloidal" location of x-point
zax	:"poloidal" location of divertor plate
alfxt	:poloidal mesh nonuniformity factor
btfix	:constant total B-field
bpolfix	:constant poloidal B-field

Within UEDGE, the variable rm gives the "radial" distance across the B-field and zm gives the poloidal distance.

For the cylindrical case, an annulus is simulated with a minimum radius of radm and a maximum radius of radx; the radial coordinate is rm. The axial distances are controlled by za0 and zax. The B-field is uniform and in the zm direction if one sets bpolfix=btfix.

5. Running UEDGE in the Time-Dependent Mode

UEDGE can use a couple of automated ODE integrators. The variable name that selects the integrator is called svrpkg, and one sets it by typing svrpkg="iname" where iname is one of the following:

```
iname = vodpk        # preconditioned Krylov package for ODE's
iname = daspk        # preconditioned Krylov package for ODE's & algebraic eqns
```

The default is svrpkg="vodpk". This ODE solvers have been developed by G. Byrne [27] and A. Hindmarsh, and is available through the NETLIB through the Web Site http://www.netlib.org/.

There are also two Newton solvers with the names nksol and newton. More details of the Newton solvers are described following the next section on time-dependent simulations. However, we have found that using the nksol option with a timestep dtreal as described below in Sec. 5.5 is most robust. The NKSOL modified Newton solver (without the dtreal timestep) has been developed by P.N. Brown and colleagues at LLNL and follows the procedures given in Ref. 28. More recent replacements for VODPK and NKSOL which run on parallel computers are PVODE and KINSOL. [29]. A version of UEDGE does run on parallel computers [30], but the operational details are not included in this manual.

It is very effective to precondition the Jacobian matrix for both the time-dependent and the steady-state Newton iterations. The options are discussed in the section on the Newton solver nksol below (search for the words nksol and premeth). Here we just mention that three options are available for the time-dependent mode, premeth="banded", "inel", or "ilut".

5.1. Setting simulation time and diagnostic output

Output data concerning the performance of the time integration is stored in a sequence of evenly spaced logarithmic time intervals. The number of outputs is set by isteps(1), which is defaulted to 100. The total simulation time is given by trange*runtim [sec]. The variable runtim gives the time increment for the first interval; trange and runtim are defaulted to 1.e+7 and 1.e-7, respectively; the default total simulation time is thus 1.0 sec. Specifically, the output time corresponding to the cumulative output index, say iout, is tout = (1.17489756)**iout * runtim. The plasma variables are also stored at these output times in the arrays nist1, upst1, test1, tist1, ngst1, and phist1 (see subroutine uedriv in file odesolve.m).

The most important thing to know from the last paragraph is that for the default settings, the total simulation time is 1.e7*runtim seconds.

5.2. Calculation of Jacobian

We use the same subroutine, pandf, to evaluate the full right-hand sides of the ODE over the whole grid, and to calculate the Jacobian. In calculating the Jacobian, the range of the do-loops over the grid is restricted to the vicinity of the variable that is being perturbed. This range is controlled by three variables: xlinc(=1) is the incremental range to smaller ix, xrinc(=2) is the incremental range to larger ix, and yinc(=1) is the incremental range to both smaller and larger iy. One can test the Jacobian calculation by setting these to values larger than nx+1 and ny+1 so that the whole range of the do-loops is done for every perturbation; this is very inefficient for normal use, however.

5.3. Accuracy

The relative accuracy of the time-dependent integration is set by rtolv, which is a vector of length 30 to allow for the possibility of setting up a maximum of 30 different sequential runs where one might change rtolv and runtim, for example; in practice, we may use 2 or 3 for grid sequencing. For a given run rtolv is used to set the **vodpk** relative error variable **rtol**. We then define the **vodpk** absolute error variable **atol(i)=catol*rtol*(guess at solution for variable i)**. Here **catol** stands for a set of scale factors for each variable set, i.e., **cniatol**, **cupatol**, **cteatol**, **ctiatol**, **cngatol**, and **cphiatol**. Typically, we choose **rtolv**=1.e-3 or 1.e-4, which is large by usual **vodpk** standards. However, we want to reach steady state as quickly as possible. The usage of the large **rtolv** required us to add a variable to the Krylov solvers for calculating the vector A*v by finite difference. The perturbation previously used for the finite difference was rtol on the assumption that the user would choose **rtol** ∼sqrt(machine roundoff) or ∼1.e-7 for the Cray. As we may have **rtol** ∼1.e-3, we let the perturbation in the Krylov solver be **srtolpk*rtol**, with the default **srtolpk**=1.e-4 used to give a perturbation of ∼1.e-7.

5.4. Boundary conditions for the time-Dependent mode

The boundary conditions are set in subroutine bouncon (see file boundary.m). For the solver **vodpk**, the boundary conditions are implemented as ODE's. For example, if we want the density **ni** to be **nb**, then the equation is

$$\frac{\partial n_i}{\partial t} = -cnurn \times nurlx(ni - nb) \tag{1}$$

where **nurlx**=1.e8 [sec] is a large relaxation frequency to force the boundary condition to be satisfied on a time scale short compared to the evolution of the non-boundary variables. The scale factor **cnurn** (=1 for default) applies only to the density equations. A similar equation is used to specify a flux-like boundary condition. The other variables have the same type of boundary equations and use the scale factors **cnuru**, **cnure**, **cnuri**, **cnurg**, and **cnurp** to allow independent adjustment for up, te, ti, ng, and phi, respectively. It should be noted that the potential equation, arising from $\nabla \cdot \mathbf{J} = 0$, has no time derivative when inertial and finite charge effects are ignored, and thus is treated this same way.

If one uses daspk as the solver, the boundary conditions are specified as algebraic equations. This guarantees that the boundary conditions are satisfied at each timestep and should be viewed as the preferable method. At the initial time, the algebraic equations must be satisfied to a given level of accuracy. This is now performed by effectively using the NKSOL Newton solver to satisfy only the boundary conditions and the potential equation if it is switched on (isphion=1).

5.5. *Using the* NKSOL *solver in a time-dependent mode*

It is possible to use svrpkg="nksol" in the time-dependent mode by setting dtreal to the desired timestep in seconds. Here a term is added to each of the non-boundary equations to account for a linear, or backward Euler, time advance; i.e., d(yl)/dt → (yl_new - yl_old)/dtreal. It is possible to also add the dtreal term to the boundary equations and the potential equation by setting the flag isbcwdt=1; this can be useful for relaxing the complete system far from equilibrium and is similar to what vodpk does. The number of such timesteps is controlled by the parameter nsteps_nk (defaulted to 1). If nsteps_nk > nsteps, the time-dependent arrays test1(istep,ix,iy) will be filled until the number of steps exceeds nsteps, and then the last value will be overwritten with the last value. This option is similar to the pseudo timestep method for NKSOL described below which is controlled by the parameter dtnewt. Note that these cannot be used together, and an error message is issued if dtreal and dtnewt are simultaneously less than 1.e5 seconds.

6. Running UEDGE with a direct Newton iteration to steady state

6.1. *Switches and diagnostic output*

To invoke the simple direct (non-Krylov) Newton iteration, set svrpkg="newton" (note that this option is now rarely used). The code then uses the subroutine newton to update the solution in the form

$$\delta yl = -J^{-1}F(yl_{old}) \tag{2}$$

where F is the right-hand side of the ODE's, δyl is the change in yl, and J is the Jacobian. At each iteration, the code prints out two lines of diagnostics. Here, sumnew1 is the average change in the magnitude of the yl's for this iteration, sumr1dy is the average of abs($\delta yl/yl$), and saux2 is the fraction of the Newton update allowed based on the variable rlx. Here rlx is the maximum amount that any yl is allowed to change relative to its old value for a given iteration. The default is rlx=0.4. The second output line gives sumf, the average of the magnitude of the right-hand-sides, ivmxchng is the ieq index of the yl(ieq) that has the largest magnitude of $\delta yl/yl$, and the (ix,iy) gives the location on the grid for this yl variable.

6.2. Preconditioning for the svrpkg="newton" *case*

Only premeth="banded" works for this option. It uses the direct banded solver SGBFA. An error message will be received if any other option is specified for premeth when svrpkg="newton".

6.3. Determining convergence trends and abort command

The Newton iteration should show a clear trend toward convergence, i.e., sumnew1 and sumf decreasing after 5 to 10 iterations. If this does not occur, experience shows that convergence is very unlikely. To abort the iteration, type ctrl-c which will return you to the DEBUG> prompt; following the instructions that appear on the terminal, you may interrogate UEDGE and then type cont to continue, or type abort to return to the UEDGE> prompt. Another good measure of convergence is the initial value of saux2; if saux2 $<$ 1.e-2, convergence is very unlikely, if 1.e-2 $<$ saux2 $<$ 1.e-1, convergence is somewhat likely, and if saux2 $>$ 1.e-1, convergence is quite likely.

The criterion for convergence is that sumnew1 $<$ rwmin, with rwmin=1.e-11 as the default. Other useful control variables are nmaxnewt which is the maximum number of iterations that the code will try (with a absolute hardwired upper limit of 101). The variable scrit causes the old Jacobian to be used if the average of abs($\delta yl/yl$) is less than scrit; the default is scrit=1.e-4.

7. Running UEDGE with Krylov-Newton Iterations to Steady State

7.1. Switches and diagnostic output

To invoke the preconditioned Krylov Newton solver option based on Peter Brown's NKSOL package, set svrpkg="nksol". The **nksol** option is generally preferred over the **newton** option. There are a variety of options for this package that are briefly described in the variable descriptor file bbb.v (groups Lsode and Ilut). Commonly changed flags are mfnksol, mdif, and mmaxu:

mfnksol	= 1	:dogleg search strategy using GMRES iterative solver
	= 2	:linesearch with Arnoldi method iterative solver
	= 3	:linesearch with GMRES method (default)
mdif	= 0	:Matrix-vector multiply J*v is approximated by numerical
		:finite difference of RHS (default)
	= 1	:Matrix-vector multiply J*v is done directly with current J
rlx	= 0.4	:restricts relative change of density and temperatures to be
		:less than rlx at each point
stepmx	=1.e9	:restricts global change of variables, sum[sqrt[(del(u)**2)]],
		:to be less than stepmx. Can be used instead of rlx.
itermx	= 30	:Maximum number of nonlinear iterations
incpset	= 5	:Maximum nonlinear iteration before Jacobian is reevaluated
mmaxu	=	:Now calculated internally with the algorithm mmaxu=neq**0.5
		:To set a specific value at input, set ismmaxuc=0 (default is 1).
epscon1	= 0.1	:Use to define tolerance of linear iterative matrix solutions
		:with the algorithm epsfac= epscon1*min(epscon2+frnm). Final
		:tolerance is epsfac*frnm
epscon2	=1.e-2	:Use to define tolerance of linear iterative matrix solutions
		:with the algorithm epsfac= epscon1*min(epscon2+frnm). Final
		:tolerance is epsfac*frnm

Note that negative values of **mfnksol** have the same meaning as the positive ones, except that the global constraints are not used (so **fnrm** can increase substantially from one iteration to another).

7.2. Preconditioning options

An important part of the Newton iteration is the preconditioning of the Jacobian matrix. There are several methods available and these are controlled by the flag premeth="iname"; the different options for the preconditioner are:

premeth="banded"	:uses the direct banded solver SGBFA. Requires a lot of :storage for large problems, but is fast on the Cray for :moderate size problems
premeth="inel"	:uses a partial LU decomposition with fill-in on existing :diagonals of Jacobian only; called ILU0
premeth="ilut"	:uses a partial LU decomposition about existing elements of :Jacobian - not based on diagonals only. Amount of fill-in controlled :by lfililut; typical problems require lfililut=3 to 100.

The output data on performance of the nksol routine at each nonlinear iteration is controlled by the flag iprint. No output occurs if iprint=0. If iprint=1 (default), the iteration count, norm of the residual, and number of right-hand-side evaluations are printed, indicating roughly the number of linear iterations. If iprint=2, detailed data concerning the linear iteration is printed for each nonlinear iteration: the norm of the residual and the value of norm required to meet convergence test. Also, if the constraint condition preventing negative densities or temperatures, or a relative step size is too big [Del(u)/u > rlx], resulting in a reduced step size, the message ivio=1, pnrm=... will appear.

7.3. Row and column scaling and rescaling

Several techniques are used to improve the numerical properties of the Jacobian preconditioning matrix and to better condition the nonlinear Newton problem. The most straightforward is scaling the rows of the Jacobian by the largest element in the row; this is effected by the switch issfon=1 (default) which generates the scaling vector sfscal. This scaling is actually applied to the Jacobian if isrnorm=1 (default).

Column scaling is a more recent addition (mid-1995) and is turned on by the switch iscolnorm (default=0). Presently, either iscolnorm = 2 or 3 is recommended, and both have nearly the same effect; 2 completely disregards the old global scaling and 3 does the column scaling after the global scaling. The column scaling is essentially a local normalization of the variables over the mesh to be of order unity, which improves the numerical solve-ability of the system.

Rescaling of the Jacobian matrix is activated by the switch ireorder=1 (default). This causes the Jacobian to be reordered using the reverse Cuthill-McKee algorithm. Typically it reduces the number of linear Krylov iterations by 30-50%, but can make the difference between convergence and no convergence.

7.4. Pseudo-transient timestep

A pseudo timestep can be added to the Jacobian for svrpkg="nksol" which generally increases the radius of convergence (one can take larger steps away from existing solutions), but decreases the rate of convergence. The timestep adds a diagonal term to the left-hand side of the linear matrix equation but not to the right-hand side. Specifically, consider the time-dependent equation for a vector of variables x of the form dx/dt+f = 0. Performing a Taylor series expansion gives the Jacobian, J, and

$$\frac{(x_n - x_o)}{dt} + \mathbf{J} \cdot \mathbf{x}_n = -f(x_o) \tag{3}$$

where x_o and x_n are the variable values at the old and new timestep, respectively. The pseudo transient technique neglects the $-x_o/dt$, but retains x_n/dt, yielding the equation

$$(\mathbf{I}/dt + \mathbf{J}) \cdot \mathbf{x}_n = -f(x_o) \tag{4}$$

where $\mathbf{I}$ is the identity matrix. This additional term is added to both the preconditioning Jacobian, and to the $\mathbf{J} \cdot \mathbf{x}$ finite-difference Jacobian-vector product calculated in the Krylov algorithm.

In UEDGE, the pseudo timestep, dt, is called dtnewt. The default value for dtnewt is 1.e20 seconds which effectively removes the I/dt term. To use this technique, it is typical to start with dtnewt=1.e-5 to 1.e-4, and run for about 10 iterations (set itermx=10). The residual as measured by fnrm should be decreasing, but do not expect convergence. Then update the "save" variables by doing a read reset, increase dtnewt by a factor of 3 or 10, and repeat the procedure. By the time you get to dtnewt=1.e-3, or so, the convergence should be accelerating, and one can often increase dtnewt by larger factors; typically, dtnewt=0.1-1.0 is almost equivalent to dtnewt=1.e20. As of yet, there is no systematic procedure coded for automatically running through the sequence described above, although Knoll and McHugh have had some success with the Switched-Evolution Relaxation (SER) method.

8. Boundary Condition Options

8.1. Specifying gas input and pumping on the side-walls

One can specify up to 10 sources (or sinks - pumping regions) on the outer wall (iy=ny) and 10 on the inner wall (iy=0) by setting the variable nwsor to the number desired. The sources come in pairs, but the inner and outer parameters can be specified independently. These gas boundary conditions are set up in subroutine walsor. Each source uses a variable called igspsori(i) or igspsoro(i) for the inner and outer wall sources, respectively that defines which gas species the source contributes to. Thus, igspsori(i)=j means that source i contributes to gas species j.

The location of the sources is set by xgasi(i) and xgaso(i) for inner (private flux) and outer walls, respectively. To set these input parameters, it is helpful to examine the arrays xfwi and xfwo which give distances along these flux surfaces. If issorlb(i)=1, the distances xgasi,o(i) [m] are measured from the inner, or left divertor plate; if issorlb(i)=0, the distances are measured from the outer, or right divertor plate. The total width of the region is given by wgasi,o(i); if igasi,o(i) $>$ 0, the source is taken to be a gas source with a cosine shape over the defined region, going to zero at the edge. If there are multiple, finite strength sources present, the net source at a given location is the sum of all of the overlapping sources.

The special setting of igasi,o(i)=0 is used to specify a pumping region with a uniform albedo defined by albdsi,o(i) over the region of the source. The albedo of the side walls are specified for each gas species separately as follows: For each source which has igasi,o(i)=0, the second variable, igspsori,o(i), defines which gas species the source defines the albedo for, just as for finite gas sources (see above). For example, if you want to set the albedoi (for the private-flux wall) for gas species 1 to 0.95 and that of gas species 2 to 0.90, you should use two sources by setting nwsor=2, and

```
igasi(1:2) = 0
xgasi(1:2) = 0
wgasi(1:2) = 1000.
igspsori(1) = 1
```

albdsi(1) = 0.95

igspsori(2) = 2

albdsi(2) = 0.90

Here wgasi is set to a large number to span the whole simulation region. The gas input depends on the number of gas species used. If only one gas species is used (ngsp=1), the current igasi,o(i) is the boundary condition for that single species in the given region and one needs to account for simultaneous puffing and pumping by reducing the net gas input.

The wall sources can also be used to redistribute the gas flux absorbed at one location by injecting it at another location. This coupling is limited to reinjecting gas from the same region (private-flux or outer-wall) from which it escapes. Here the neutral flux is calculated from the albedo defined over a specified source region, and then reinjecting as a gas flux over a different (or the same) region with a cosine distribution characteristic of the sources. As an example, consider the current produced by the albedo albdsi(k) over the region specified by source k with location and width xgasi(k) and wgasi(k), respectively. To reinject this current over the region specified by source j, with location and width xgasi(j) and wgasi(j), set ncpli(k)=j to establish the coupling, and set cplsori(k) equal to the fraction of the current that will come through the source with index j [cplsor(k)=1 gives the full current at source j]. Note that one may use k=j.

One can also specify the side walls as material surfaces that emit recycled gas. This is done by switching on the flag matwsi,o(i)=1 for a given wall source with igasi,o(i)=0.; if igasi,o(i) is nonzero, the gas input boundary conditions will be used and not the material wall condition even if matwsi,o(i)=1. The gas input at the boundary for matwsi,o(i)=1 is determined by the wall recycling coefficients recycw(1) for hydrogen. The models for material side walls are still evolving, so users should check with the authors if they intend to use this option.

The end plates are also sources of neutrals, where the gas flux is specified as -recycp*(ion flux). Again, recycp(1) refers to the flux into the hydrogen gas, and recycp(2...) refers to impurity gases.

8.2. Other side-wall boundary condition options

There are several options to use for the density and temperature boundary conditions on the private-flux wall (iy=0) and the outer wall (iy=ny+1).

Constant value (Dirichlet) boundary conditions:

For ion density, set isnwconi=1 for the private flux boundary (iy=0) and isnwcono=1 for the outer wall boundary (iy=ny+1). The density values can be set through the poloidal arrays nwalli and nwallo for the inner and outer walls, respectively. Be careful if the mesh is increased and interpolation is used, as one must then update nwalli and nwallo.

For electron and ion temperature, set istepfc=1. and istipfc=1. for the private-flux boundary, and set istewc=1. and istiwc=1. for the outer wall boundary. Each of these switches can be set separately. The temperatures are then set to tedge (eV) or can be given separate and poloidally varying values through the arrays tewalli, tiwalli, tewallo, and tiwallo, where the final letter denotes inner(i) or private-flux boundary and outer(o), or outer wall boundary. Before you set tewalli, etc. to nonzero values, you must allocate memory for these arrays by typing allocate at the basis prompt. In using this option, be careful of interpolating to a larger grid, as the tewalli, etc will have to be manually interpolated after the allocate.

Flux (Neumann) boundary conditions:

For ion density, be sure that isnwconi and/or isnwcono = 0, as well as isextrnpf and/or isextrnw = 0 and ifluxni=1. Because there is a possibility of conflict between the switches, the order of dominance is as follows: isextrnpf, isnwconi, and finally ifluxni. The same is true for isextrnw, isnwcono, and ifluxni.

For electron and ion temperature, set istepfc=0., istipfc=0., istewc=0., and istiwc=0. This results in the normal derivative being set to zero, dT/dy=0. Note that the switches istepfc, istipfc, istewc, and istiwc are real variables that take on any value. Thus, you can continuously evolve from one boundary condition to the other by taking fractional steps between 0. and 1.0. This can be especially useful for Newton iterations.

Extrapolation boundary conditions:

For the density and temperatures, it is possible to set extrapolation boundary conditions at iy=0 and iy=ny+1. This condition sets the boundary value to be a linear extrapolation of the previous two points in the radial direction. For the density, the inner (private flux) and outer wall switches are isextrnpf and isexrtnw, respectively, and setting either to 1 forces the extrapolation condition independent of the settings of the other switches. For the temperature, both the electron and ion values are set together for the inside or outside with variables isextrtpf and isextrtw. Again, if the extrapolation switches are set to 1, they take precedence over any setting of istewc, etc.

8.3. End-plate boundary condition options

The ion parallel velocity is taken to be the sound speed multiplied by the user-set scale factors csin and csout at the inner and outer divertor plates, respectively. If the switch isupss=1, then the parallel velocity is allowed to be supersonic at the plate if the solution seeks this state.

The recycling coefficients at the plates can be made a function of radial position or an albedo can be specified over a limited region to pump gas through the plate. The following variables are used to set the recycling and albedos:

ndati(igsp)	# number of data points along plate; if=0, recycling
	# is uniform and specified by recycp*recycfi
ndato(igsp)	# number of data points along outer plate; if=0,
	# recycling uniform and specified by recycp*recycfo
ydati(igsp,idat)	# dis. from inner sep. of data point for rdati & albpi
rdati(igsp,idat)	# value of recycling coeff. at plate location ydati;
	# in between data points, linear interp. is used
adati(igsp,idat)	# value of albedo at data point ydati; lin. interp used
ydato(igsp,idat)	# outer-plate counterpart to ydati
rdato(igsp,idat)	# outer-plate counterpart to rdati
adato(igsp,idat)	# outer-plate counterpart to adati

If the albedo in any segment along the plate is less than unity (the default), then the albedo boundary condition for the gas takes precedent over the recycling boundary condition. Note that it only makes sense to use at least two data points on the inside or outside because linear interpolation is used between the points. Operationally, one needs to first generate the mesh and run through nphygeo (just do a very short calculation) to generate the mesh locations relative to the separatrix on the plates; these are yylb and yyrb for the inner plate (left boundary) and outer plate (right boundary), respectively. With this information, you can decide where to put your data points (ydati and ydato).

8.4. Boundary conditions at the core interface

The ion density boundary condition is controlled by the variable isnicore(ifld), where ifld is the index of the ion density, ni(ifld). Thus, the settings of isnicore correspond to:

isnicore=1: set uniform, fixed density, ncore

isnicore=0: set flux to curcore

isnicore=2: set flux & ni over range

isnicore=3: set integrated flux, const ni

isnicore=4: use impur. source terms (impur only)

The gas density boundary condition is controlled by the variable isngcore(igsp), where igsp is the index of the gas species, ng(isgp). Note that inertial neutral gas is controlled by isngcore(1). Thus, the settings of isngcore correspond to:

isngcore=0: set zero flux

isngcore=1: set uniform, fixed density, ngcore

isngcore=2: set rad. grad. to sqrt(lam_i*lam_cx)

isngcore=3: extrap. for diff. gas only

isngcore=anything else: set zero deriv which was prev default for inertial hydrogen

In restarting from a isnicore=0 case (zero flux), use mfnksol=-3 if svrpkg="nksol". For the core temperature boundary conditions on Te and Ti, one may set either a specified power for electrons and ions as pcoree,i in Watts and setting the switch iflcore=1. To use fixed temperature boundary conditions, set iflcore=0, and then tcoree and tcorei give the electron and ion temperatures on the boundary in eV, respectively. The parallel velocity boundary condition is set by isupcore, where

isupcore = 0: set up=0 on core edge

isupcore = 1: set d(up)/dy=0 on core edge

isupcore = 2: set uu=0 on core edge, where uu is poloidal velocity.

8.5. Sputtering boundary conditions for the gas species

There can be wall or plate sources of gas that arise from sputtering, either physical or chemical. These are controlled by two flags for each gas species, isph_sput(igsp) for physical sputtering and isch_sput(igsp) for chemical sputtering.

8.5.1. physical sputtering on plates

We consider two settings for the switch isph_sput(igsp).

For isph_sput(igsp) = 0:

The sputtering of gas species igsp is controlled by the simple yield factor sputtr if the sputtering arrays sputto(iy,igsp) and sputti(iy,igsp) are zero as at the initialization of a run. Then sputto,i $\leftarrow$ sputtr. However, to change the sputtering after it has been set initially by sputtr, you must directly change the arrays sputti(iy,igsp) for the inner plate and sputto(iy,igsp) for the outer plate.

For isph_sput(igsp) = 1:

This setting uses the DIVIMP/JET model obtained from David Elder. To use this properly, you must set the following input parameters:

```
cion        # atomic number of the target material; default is 6 for carbon
cizb        # max charge state of plasma ions; default is 1 for hydrogen
crmb        # mass of plasma ions in AMU; default is 2. for deuterium
```

The resulting yield along the divertor plate is put into the arrays sputti(iy,igsp) and sputto(iy,igsp).

8.5.2. chemical sputtering on side walls

Similarly, on the side wall, we use the switch isch_sput, but it now has more than just two options.

For isch_sput(igsp) = 0:

This setting allows the user to specify the chemical sputtering yield by initializing chemsputi,o(i,j), where the resulting flux boundary condition for gas species i is then

```
fngy(igsp=i) = Sum_j [chemsputi,o(i,j)*ng(j)*vt*sy]
```

For isch_sput(igsp) > 0:

Note that isch_sput(igsp) should be nonzero for only one gas species and this species should be carbon. These settings (1-7) use various models for the chemical sputtering of carbon from the side walls; this package comes from DIVIMP via David Elder (U. Toronto, private comm., 1998). The various models are:

isch_sput	Options for chemical sputtering:
1	Garcia-Rosales' formula (EPS94)
2	according to Pospieszczyk (EPS95)
3	Vietzke (in Phys. Processes of Interaction Fusion Plasma with Solids
4	Haasz (Submitted to J.Nucl.Mater.,Dec. 1995)
5	Roth & Garcia-Rosales (Nucl. Fusion, March 1996)
6	Haasz 1997 (Brian Mech's PhD Thesis)
7	Haasz 1997 + reduced 1/5 from 10->5 eV (Porter)

It is recommended that isch_sput = 5 or 6 be used, although recently G.D. Porter has a new fit (7) which departs from (6) at low energy, and is a better fit to the (Haasz) data at the low energy. Other input is the temperature of the surface, t_surf, in degrees K; the default is 300 K. The resulting chemical sputtering yield is stored in the arrays yld_carbi,o(ix) along the inner and outer walls.

9. Sources and Sinks

It is possible to specify fixed particle, current and energy sources having specific locations and Gaussian widths through the arrays volpsor, voljcsor, pwrsore, and pwrsori. These are part of the variable group Volsrc, and various control parameters are briefly described in the variable descriptor file, bbb.v.

The sources and sinks are normally determined by ionization of neutral gas and recombination of ion-electrons into neutrals. A special background source is used to prevent the neutral density from becoming too small. The gas continuity has the form

$$\frac{\partial ng}{\partial t} + \nabla \cdot (n_g \mathbf{v}_g) = -nuiz(n_g - bgsor) \qquad (5)$$

where $n_g = ng$ is the gas density, and bgsor is given by bgsor = ngbackg*(0.9 + 0.1*(ngbackg/ng)**ingb). Normally, nbackg=1.e15 m**(-3), and ingb=0 for defaults. However, sometimes it is useful to set ingb=2 or larger to prevent "pump out" of low density cells.

10. Flux-Limiting Transport Coefficients

The flux limits used in UEDGE for the parallel transport are of the form

$$\chi = \chi_s/[1 + |q_s/q_f|^{flgam}]^{1/flgam} \tag{6}$$

where χ_s is a classical (Spitzer) diffusion transport coefficient, and flgam=1 is the default value (seldom changed). The second heat flux, q_f is the free-streaming flux defined by $q_f = flalfe * ne * v_{te} * te$, where flalfe is a parameter often set to 0.21 to match some kinetic modeling, ne is the electron density, vte=sqrt(te/me), and te is the electron temperature. If flalfe=1e20 (the default), the flux limiting is effectively switched off. There are three flux limits used for the plasma:

flafle (recommend=0.21)	:for electron parallel heat flux
flafli (recommend=0.21)	:for ion parallel heat flux
flaflv (recommend=1.0)	:for ion parallel viscosity

These parameters are all defaulted to large values (1e20 or 1e10) which gives effectively no flux limiting. The recommended values are obtained from fits to Monte Carlo and Fokker Planck calculations, but are not universal.

There are also flux-limiting coefficients for the diffusive gas fluxes. These are called flalfgx, flalfgy, and flalfgxy for diffusion driven by the density gradients in the x-, y-, and nonorthogonal xy- directions. In addition, fluxes can be driven by gradients in temperature which have flux-limit coefficients flalftgx and flalftgy. Finally, the neutral gas viscosity coefficients can be limited through the parameters flalfvgx and flalfvgy. All of the gas flux-limit parameters are defaulted to large numbers, but physically reasonable values are in the range of unity which are left as an option for the user. Our experience is that the code can have difficulty with these flux limits if gradients become too steep - this is the reason for having them switched off as a default.

11. Models for Neutral Gas

11.1. *Fluid and diffusive models for atoms*

UEDGE uses two models for the neutral gas, the most general being a fluid model that solves the parallel momentum equation along the direction of the magnetic field,B, and diffusion in the two directions perpendicular to B (implemented by F. Wising). For this model, set isupgon(1)=1, isngon(1)=0, nhsp=2, and ziin(2)=0. Neutral viscosity and thermal conduction is included using both charge-exchange collisions and neutral-neutral collisions.

The simpler gas model solves a diffusion equation in the 2-D poloidal plane. For hydrogen, it is activated by setting isngon(1)=1, isupgon(1)=0, and nhsp=1. The diffusion coefficient is given by D_g=Ti/[mi*(nucx+nuiz)].

The impurity gas is modeled by the diffusion equation just mentioned and is activated by setting isngon(2)=1 (if more than one impurity species is present, then isngon(3)=1, etc.). Here the diffusion coefficient is somewhat more general by including elastic collisions as D_g=Tg/(mg*nuix), where

nuix = rcxighg*nucx_h + nuiz + massfac*(kelighi*ni_h + kelighg*ng_h)

Here, nucx_h is the charge-exchange frequency between hydrogen neutrals and impurity ions, rcxighg is a scale factor (usually small) to convert this rate to that between impurity neutrals and hydrogen ions. To account for elastic collisions of impurity gas with hydrogen ions and gas, we use the last two terms. Details of this model were suggested by S. Krasheninnikov, where

massfac = 16*mi_h / [3*(mi_imp + mi_h)]

and kelighi and kelighg are the ⟨sigma-v⟩rates. Estimated values are kelighi = kelighg = 5e-16 m**3/s at temperatures of $\sim$ 1 eV, but the temperature dependence is neglected. The precise values are uncertain. Values of kelighi and kelighg should be set in the users input file. Thus, if you set rcxighg=0., you should set kelighi(igsp) = kelighg(igsp) = 5.e-16, or some more accurate value if available. This procedure prevents D_g from becoming very large in low temperature ($\sim$ 1 eV) regions when rcxighg=0 and nuiz are very small.

11.2. Options for temperature of neutrals

One can let the neutrals have a multiple of the common ion temperature or use a specified constant value. These options are controlled as follows:

```
If istgcon=0, then
      the gas temperature, tg, is a multiple of the ion temperature, ti.
      Specifically, then tg=rtg2ti*ti across the whole mesh.
If istgcon=1, then
      tg = tgas*ev, where tgas is an input variable in eV, and tg has this
      same constant value across the mesh.
```

11.3. Inclusion of fluid molecules

A fluid component can be used to represent the molecules which evolve from the wall to describe the thermal desorption phase of recycling; this is usually the dominant recycling channel.

To include hydrogen molecules, one should set the following input parameters:

```
ngsp = 2           # if no impurities are present
nhgsp = 2          # tells code that two hydrogen gas species are present
ishymol = 1        # switch to turn on hydrogen molecules
recycp(1)= -0.5    # neg. recycp(1) acts like -albedo for atomic gas
recycp(2)= 0.98    # recycling into molecular channel for ions + atoms
recycw(1)= -0.5    # neg. recycw(1) acts like -albedo for atomic gas
recycw(2)= 0.98    # recycling into molecular channel for ions + atoms
cdifg(2) = 0.05    # reduces mol gas diff coeff to simulate wall temp
```

Note that recycp applies to the divertor plates and recycw applies to the side walls when matwsi,o > 0. The second gas species then corresponds to the molecules while the first is the atomic species. The atomic species can be either diffusive neutrals or 1-D Navier-Stokes as described just above. The diffusion coefficient for the molecular gas is D_g=cdifg(2)*Tg/(mg*nuix), where nuix is now given by

```
nuix = nu_diss + massfac*(kelighi*ni_h + kelighg*ng_h)
```

where nu_diss is the dissociation rate calculated using a polynomial fit obtained from the EIRENE neutral Monte Carlo code and massfac is defined in the previous section.

11.4. Coupling to Monte Carlo neutral codes

The hydrogenic fluid neutrals model can be turned off and replaced by a Monte Carlo neutrals model. In the simplest scheme, one uses a numerically explicit time-dependent coupling of plasma and neutrals models, with the models communicating via disk files. The UEDGE plasma mesh information is written to a disk file, fort.30, with the command:

```
call write30("fort.30", runid)
```

The UEDGE plasma background information, i.e., density, temperatures and flow velocity, is written to a disk file, fort.31, with the command:

```
call write31("fort.31", runid)
```

where runid is some header text to identify the run.

The hydrogenic fluid neutrals model is turned off and the Monte Carlo neutrals on via the switches:

```
nhsp=1
isupgon(1)=0
isngon(1)=0
ismcnon=1
```

which turns off the hydrogenic fluid neutrals contributions to the plasma source terms for ion density, ion parallel momentum, electron temperature and ion temperature. The user must replace these sources with corresponding sources from the Monte Carlo neutrals model at each time step before executing the plasma model with exmain. For the EIRENE Monte Carlo code, the procedure is as follows:

1. call read32("fort.32"); this reads a data file, fort.32, which contains normalized plasma source terms due to each 'stratum' in EIRENE; the data arrays are sni, smo, see, sei and the normalization constant(s) wsor.

2. convert from normalized source terms to physical source terms, e.g., mcnsor_ni = -wsor*sni

3. convert the source for total ion energy to a source for thermal ion energy only, i.e., mcnsor_ti = mcnsor_ti - up*mcnsor_up + (.5*mi*up**2)*mcnsor_ni

4. compute total plasma sources by summing over all 'strata', e.g., uesor_ni(ix,iy,ifld) = sum on istra [mcnsor_ni(ix,iy,ifld,istra)]

After executing the plasma model, one writes the plasma background data for the next Monte Carlo neutrals calculation with a call to subroutine write31 as noted above.

The Monte Carlo code can be executed from within UEDGE via the parser command:

```
basisexe("eirobjx < input.dat > eir.log")
```

where eirobjx is the name of the executable and input.dat and eir.log are standard input and output files for the EIRENE code. Some diagnostic output from the EIRENE code is accessible within UEDGE with the parser command:

```
call read44("fort.44")
```

In particular, the atomic neutral density in the array naf(1:nx,1:ny,1) may be compared with the fluid model result inng(1:nx,1:ny,1). A similar procedure is followed for coupling to the DEGAS2 Monte Carlo code [31].

12. Models for Hydrogen Ionization, Radiation, and Recombination

The calculation of the ionization, radiation, and recombination terms in the ion and gas continuity equations is taken either from an analytic model or linear interpolation of data from table look-up. For most of the options, recombination is switched on by isrecmon=1. The model used is controlled by the variable istabon and has the following options:

istabon=0	:Analytic model for ionization (from Braams in B2) and
	:charge-exchange; no recombination
istabon=1	:Tables from ADPAK by Hulse via Braams; not recommended
	:by Doug Post for the low temperature (< 50 eV) regime
istabon=2	:Tables from STAHL by Behringer at Garching via Braams
istabon=3	:Tables used in DEGAS from Janev, Post, et al., 1984
istabon=4	:Extended-DEGAS tables from Doug Post '93 for temperatures
	:down to .063 eV and densities up to 1.0e+23 /m**3;
	:linear interpolation is done for rsa vs log Te and log ne;
	:analytic model for charge-exchange.
istabon=5	:Same extended-DEGAS tables as option 4, but with spline
	:interpolation for log10(rsa) vs log(te) and log10(ne)
istabon=6	:Same extended-DEGAS tables as option 4, but with spline
	:interpolation for rsa vs log(te) and log10(ne)
istabon=7	:Campbell's polynomial fit for rsa vs log10(te) and log10(ne)
istabon=8	:New DEGAS tables from D. Stotler Oct '93; separate electron
	:radiation loss rates due to ionization and recombination;
	:linear interpolation as in option 4; radiative loss rates
	:more accurate than option 4 for low Te and/or large ne.
istabon=9	:from Stotler at PPPL (95/07/10) using log(Te)-sigv
istabon=10	:from Stotler at PPPL (95/07/10) using log(Te)-log(sigv)

The presently preferred table is istabon=10 which is a more complete table from that reported in Ref. 32.

One can find the value of various rate parameters by calling the appropriate function from the BASIS parser with specific arguments (not arrays). These are functions for ⟨sigma*v⟩and are as follows:

rsa(te(ix,iy),ne(ix,iy),0)	:⟨sigma*v⟩_ionization [m**3/s]
rcx(ti(ix,iy),ni(ix,iy),1)	:⟨sigma*v⟩_charge_exchange [m**3/s]
rra(te(ix,iy),ne(ix,iy),1)	:⟨sigma*⟩_recombination [m**3/s]

The radiative loss rates associated with ionization and recombination can be obtained by calling the following functions:

erl1(te(ix,iy),ne(ix,iy))	:ne*⟨sigma*v*E_rad⟩_ioniz [J/s]
erl2(te(ix,iy),ne(ix,iy))	:ne*⟨sigma*v*E_rad⟩_recom [J/s]

The calculated collision frequencies on the 2-D mesh are stored and can be viewed in the following variables:

nuiz(ix,iy)	:ionization frequency [1/s]
nucx(ix,iy)	:charge-exchange frequency [1/s]
nurc(ix,iy)	:recombination frequency [1/s]; need isrecmon=1

The total (radiation + 13.6*ev) electron energy loss per ionization on the 2-d mesh is stored and can be viewed in the following variable:

eeli(ix,iy)	:energy loss per ionization [Joules]

13. Model for Impurity Radiation and Transport

13.1. Fixed-fraction model

The impurity radiation for the fixed-fraction model uses a look-up table based on non-equilibrium coronal results from the MIST code (by R. Hulse and D. Post) for a given impurity. The impurity emissivity depends on electron temperature, charge-exchange recombination on neutral hydrogen and impurity lifetime due to convection. The impurity charge-exchange rate is evaluated using the neutral hydrogen density calculated by UEDGE, whereas the impurity lifetime is presently specified by the user in the 2-D array atau(ix,iy) [sec]. The impurity density is determined as a fraction of ne(ix,iy) by the user-specified array afrac(ix,iy), i.e., the impurity density = ne*afrac.

The impurity radiation is removed from the electron energy equation by setting the switch isimpon > 0. [For isimpon=1, you must explicitly read a set of impurity radiation data files with the command read setup.nitrogen; this option is now obsolete]. For isimpon=2, impurity radiation data files are automatically read when the code is executed with the exmain command; in this case the code looks for a file in your working directory with the default name mist.dat.

One typically specifies a constant fraction of impurities by typing afrac(,)=0.01 for 1%; the impurity lifetime atau(,) is defaulted to 1 second, so it does not affect the calculation significantly unless the user sets a lower value. We are working to improve the impurity model so that the fraction and lifetime are calculated self-consistently from transport.

13.2. Multi-species models

One can also treat the impurities by following the densities of the individual charge states. This is done by setting isimpon=5 where the FMOMBAL package by Steve Hirshman, ORNL, is used to calculate the friction forces between species and a mass-averaged momentum equation is solved for all the species. Another option is to set isimpon=6 where the friction forces are determined from analytic formulae and the individual impurity parallel velocities come from the force balance equation that results when impurity inertia and viscosity are ignored. It is possible to solve individual parallel momentum equations for each charge state. This requires setting the parameter nusp_imp to the number of impurity momentum equations. The radial transport of the impurity species is determined by the diffusion coefficients, difni, which may be set differently for each charge state followed (see the section on anomalous transport).

The look-up tables for impurity ionization, radiation, and recombination rates come from a computer code developed by B. Braams which writes out tables for either ADPAK rates [33] or STRAHL rates [34]. Which rates are used are controlled by the character variable mcfilename; its use is illustrated in the multiple-isotope example shown below.

Multiple isotopes can be followed simultaneously. Here the impurity gas is modeled using an diffusion equation. The relevant parameters for a typical input file with helium and neon are as follows (this cases assumes the first two “ion” species are hydrogen ions and hydrogen neutrals using the inertial neutral model; the impurities thus start with ion species 3, but gas species 2):

```
# Impurities
    isimpon =6           #Use force balance equation
# Impurity gas
    ngsp = 3             #total number of gas species (hydrogen+ impurities)
    isngon(2:3) = 1      #turn on impurity gas equations
    recycp(2:3) = 1.0    #plate recycling coeff of helium and neon
```

```
# Helium species
        nzsp(1) = 2             #number of helium charge states used
        minu(3:4) = 4.          #helium mass in AMU
        ziin(3:4) = iota(1:2)   #charge for each state used
        znuclin(3:4) = 2        #nuclear charge for helium
        isnicore(4)= 1          # =1 for fixed core density of He++
        ncore(4)=2.e18          #density of He++ at core boundary
# Neon species
        nzsp(2) = 8             #number of neon charge states used
        minu(5:12) = 20.        #neon mass in AMU
        ziin(5:12) = iota(1:8)#charge for each state used
        znuclin(5:12) = 10      #nuclear charge
        isnicore(12)= 1         # =1 for fixed core density of Ne+8
        ncore(12)=4.e17         #density of Ne+8 at core boundary
# Specify impurity data files
        nzdf = 2                #number of impurity data files to be read
        mcfilename = ["He_rates.strahl","Ne_rates.strahl"] #data file names
```

14. Specifying Anomalous Radial Transport Coefficients

The simplest description of radial transport is a set of spatially constant diffusion coefficients for density, parallel momentum, electron energy, and ion energy. All are in units of m**2/s, and the density and momentum coefficients allow 12 (expandable) locations for 12 species. The coefficients are as follows:

```
difni(i)          # radial (or y-direction) density diffusion
vcony(i)          # radial convective pinch velocity
difni2(i)         # perpendicular density diffusion in "2" direction in
                  # flux surface (perp. to y and —— directions)
travis(i)         # radial parallel momentum diffusion
difutm(i)         # radial toroidal momentum diffusion (for potential eqn)
```

kye	# radial electron energy diffusion
kyi	# radial ion energy diffusion

In addition, one can introduce user-specified, spatially-dependent diffusion coefficients, or let the code calculate Bohm-like diffusion coefficient which are added to the ones above. The switch isbohmcalc determines which of these two options is active:

isbohmcalc = 1 → code fills 2-D arrays dif_use, tra_use, kye_use, and kyi_use = Te/(16eB)
(default=1) (these Bohm rates are calc. on x,y-mesh-faces)

isbohmcalc = 0 → code uses whatever user initially sets for arrays dif_use, dif2_use,
tra_use, kye_use, and kyi_use. In addition, vy_use may be
set to a user-specified array as the pinch velocity.

The net diffusion is defined by using scale factors of the density, electron energy, and ion energy separately. The net diffusion coefficients for the isbohmcalc=1 case are thus

difni	→ difni + facbni *dif_use(ix,iy)
difni2	→ difni2 + facbni2*dif2_use(ix,iy)
travis	→ travis + facbup*tra_use(ix,iy)
kye	→ kye + facbee *kye_use(ix,iy)
kyi	→ kyi + facbei *kyi_use(ix,iy)
vy	→ vy + vy_use(ix,iy) Only if isbohmcalc=0

Nearly the same relation holds for isbohmcalc=0, except that all of the "facb..." factors are unity. Note that difni2 and dif2_use typically add small contributions to the poloidal transport that is dominated by the projection of the parallel transport; we thus usually leavedifni2 = facbni2 = dif2_use = 0. Here facbni, etc. are only scalars, thus applying equally to all ion species. If one wishes to use only user-specified diffusion coefficients, be sure to set difni, difni2, kye, and kyi to zero as all but difni2 are defaulted to unity if they are not set in the input file. Also, if the mesh size changes and the user is specifying the values of dif_use, etc. (isbohmcalc=0, facbni=1.), the arrays dif_use, etc. must be refilled after an allocate for the new mesh size is done.

15. Converting solutions from Full-Space to Half-Space & Vice Versa

It is convenient to use the ability of BASIS to manipulate arrays to convert a full-space solution to a half-space solution. Here we show how to do this for a single-null solution and for a lower double-null solution. Below this is a BASIS script that does the reverse, i.e., takes a half-space solution and symmetrizes it as the initial guess of a full-space solution. Here a full-space problem is one with two divertor plates, one at each end of the x (poloidal) domain, and a half-space problem is one where on end of the x domain is a symmetry plane. Generally, the symmetry plane is at the left boundary (isfixlb=2), but can also be at the right boundary (isfixrb=2).

15.1. Converting from full-space to half-space

Note that such conversions take place most straightforwardly if one uses fixed temperature core boundary conditions as the input power does not then need to be adjusted. If you are running with power boundary conditions (iflcore=1), first determine the core temperatures, set tcoree and tcorei to these and change to iflcore=0, and then do the conversion to a different sized space.

First, the bottom double-null case:

(Lines beginning with # are comments and may be omitted. Also, the letter variables are chameleon variables in BASIS that take on the properties of the variable to which they are set)

save original solution in temporary storage:

```
$n=nis;$u=ups;$e=tes;$i=tis;$g=ngs;$p=phis
```

set switches to do outer quadrant only:

```
isfixlb=2
```

for a single-null case, set

```
nxomit=nxleg(1,1)+nxcore(1,1)
```

or, for a lower double-null case, set

```
nxomit=nxleg(1,1)+nxcore(1,1)+1
```

get very crude index-interpolated solution for outer quadrant only # NOTE: do not try to converge from this state: `real oldftol=ftol; ftol=1e10; exmain; ftol=oldftol` # copy original solution to restart arrays:

```
nis=$n(nxomit:nxm+1,,)
ups=$u(nxomit:nxm+1,,)
tes=$e(nxomit:nxm+1,)
tis=$i(nxomit:nxm+1,)
ngs=$g(nxomit:nxm+1,,)
phis=$p(nxomit:nxm+1,)
```

For a single-null case:

Same as above, but also set the left-hand symmetry boundary conditions:

```
nis(nxomit,,) = nis(nxomit+1,,)
ups(nxomit,,) = 0.
tes(nxomit,) = tes(nxomit+1,)
tis(nxomit,) = tis(nxomit+1,)
ngs(nxomit,,) = ngs(nxomit+1,,)
phis(nxomit,) = phis(nxomit+1,)
```

```
exmain        # outer-quadrant-only solution, on original mesh; should converge easily
read doublep  # script to double nxleg, nxcore, nycore, nysol
exmain        # run and hopefully converge on doubled poloidal mesh
```

15.2. Converting from a half-space to a full-space

Assumes that `nxleg(1,1)=nxleg(1,2)` and `nxcore(1,1)=nxcore(1,2)`. Unlike the previous example, this does not use chameleon variables, but could

Set up needed “ss” work arrays:

```
real8 niss(0:2*nx+1,0:ny+1,1:nisp); real8 upss(0:2*nx+1,0:ny+1,1:nusp)
real8 tess(0:2*nx+1,0:ny+1); real8 tiss(0:2*nx+1,0:ny+1)
```

```
real8 ngss(0:2*nx+1,0:ny+1,1:ngsp); real8 phiss(0:2*nx+1,0:ny+1)
```

Fill work arrays:

```
do ix = 0, nx
    niss(ix,,1:nisp) = nis(nx+1-ix,,1:nisp)
    upss(ix,,1:nusp) = -ups(nx-ix,,1:nusp)
    tess(ix,) = tes(nx+1-ix,)
    tiss(ix,) = tis(nx+1-ix,)
    ngss(ix,,1:ngsp) = ngs(nx+1-ix,,1:ngsp)
    phiss(ix,) = phis(nx+1-ix,)
enddo
do ix = nx+1, 2*nx+1
    niss(ix,,1:nisp) = nis(ix-nx,,1:nisp)
    upss(ix,,1:nusp) = ups(ix-nx,,1:nusp)
    tess(ix,) = tes(ix-nx,)
    tiss(ix,) = tis(ix-nx,)
    ngss(ix,,1:ngsp) = ngs(ix-nx,,1:ngsp)
    phiss(ix,) = phis(ix-nx,)
enddo
```

Modify switches, allocate proper space for save variables and fill them.

```
nxomit = 0
isfixlb = 0
allocate
    nis = niss
    ups = upss
    tes = tess
    tis = tiss
    ngs = ngss
    phis = phiss
```

At this point, you may begin the run with an exmain command, or save the “s” variables for a restart by creating a PFB save-file, e.g.,

```
create pf_somename;write nis,ups,tes,tis,ngs,phis;close
```

ACKNOWLEDGMENTS

We are pleased to acknowledge the contributions of many people to various aspects of the UEDGE code: J.L. Milovich, F. Wising, P.N. Brown, A.C. Hindmarsh, R.B. Campbell, S.P. Hirshman, D.A. Knoll, S.I. Krasheninnikov, G.D. Porter, B.J. Braams, D. Elder, T.B. Kaiser, L.L. LoDestro, D. Stotler, and A.G. Taylor. We have also benefited from numerous discussions with R.H. Cohen, A.S. Kukushkin, D.E. Post, D.D. Ryutov, and X.Q. Xu.

Appendix A. Equations used for the UEDGE code

This section closely parallels the discussion in Ref. 25, and the interested reader may find it helpful to consult that paper for some more details. The basic form of the transport equations, without cross-field drifts, corresponds to that implemented in a number of edge transport codes being based on classical fluid equations, one of the first being the B2 [3] code as later specified in the published Ref. 4. Here we show the modifications to this equation set as used in UEDGE when classical cross-field drifts are included. The plasma velocities in our equations are denoted by the symbol **u** and differ from the total velocities **v** by having the classical cross-field pressure or temperature gradient terms omitted since these have zero divergence, or cancel with gyro-viscous terms, and therefore do not contribute to the transport. (*e.g.*, see Ref. 2. Note that $v_{||} = u_{||}$. For the poloidal ion velocity

$$u_{ix} = \frac{B_x}{B} v_{i||} + v_{x,E} + v_{ix,\nabla B}, \tag{A1}$$

where the x-component of $v_{i,\nabla B}$ can be found in Ref. 25. For the radial ion velocity

$$u_{iy} = -\frac{D_a}{n_i}\frac{\partial n_i}{\partial y} + v_{y,E} + v_{iy,vis} + v_{iy,\nabla B}, \tag{A2}$$

where the last two terms come from the corrections to $v_{i,y1}$ given in Ref. 25.

The electron velocities are obtained from

$$\mathbf{u}_e = \frac{n_i Z_i \mathbf{u}_i}{n_e} - \frac{(\mathbf{J}_{||} + \mathbf{J}_{vis} + \mathbf{J}_{\nabla B})}{e n_e} \tag{A3}$$

where again the **J**'s are only the currents with finite divergence.

The ion continuity equation is

$$\frac{\partial}{\partial t} n_i + \frac{1}{V}\frac{\partial}{\partial x}\left(\frac{V}{h_x} n_i u_{ix}\right) + \frac{1}{V}\frac{\partial}{\partial y}\left(\frac{V}{h_y} n_i u_{iy}\right) = (\langle \sigma_i v_e\rangle - \langle \sigma_r v_e \rangle) n_e n_n \tag{A4}$$

The terms $\langle \sigma_r v_e \rangle$ and $\langle \sigma_i v_e \rangle$ are rate coefficients for recombination and ionization, respectively. The metric coefficients are $h_x \equiv 1/||\nabla x||$, $h_y \equiv 1/||\nabla y||$, and $V = 2\pi R h_x h_y$ is the volume element for toroidal geometry with major radius R. [4] For brevity of presentation, the metric coefficients are suppressed in the remaining equations.

The ion parallel momentum equation is

$$\frac{\partial}{\partial t}(m_i n_i u_{i||}) + \frac{\partial}{\partial x}\left(m_i n_i u_{i||} u_{ix} - \eta_{ix}\frac{\partial v_{i||}}{\partial x}\right) + \frac{\partial}{\partial y}\left(m_i n_i v_{i||} u_{iy} - \eta_{iy}\frac{\partial v_{i||}}{\partial y}\right) = \frac{B_x}{B}\left(-\frac{\partial P_p}{\partial x}\right) \tag{A5}$$

where $P_p = P_e + P_i$, $\eta_{ix} = (B_x/B)^2\eta_{||}$ is the classical viscosity, $\eta_{iy} = m_i n \Upsilon_{a||}$ is anomalous. All classical viscosities and thermal conductivities are flux-limited to prevent unphysically large values in regions with long mean-free paths. Expressions for the classical terms can be obtained from Ref. 1.

The electron energy equation is

$$\begin{aligned}\frac{\partial}{\partial t}\left(\frac{3}{2}n_e T_e\right) + \frac{\partial}{\partial x}\left[\frac{5}{2}n_e u_{ex} T_e - \kappa_{ex}\frac{\partial T_e}{\partial x} - 0.71 n_e T_e \frac{B_x}{B}\frac{J_{||}}{e n_e}\right] + \frac{\partial}{\partial y}\left(\frac{5}{2}n_e u_{ey} T_e - \kappa_{ey}\frac{\partial T_e}{\partial y}\right) \\ = u_{ix}\frac{\partial P_e}{\partial x} - u_{iy}\frac{\partial P_i}{\partial y} - u_{iw}\frac{B_x}{B}\frac{\partial P_p}{\partial x} + \mathbf{E}\cdot\mathbf{J} - K_q(T_e - T_i) + S_{Ee}.\end{aligned} \tag{A6}$$

Here the poloidal heat conductivity is classical, $\kappa_{ex} = (B_x/B)^2\kappa_{||}$, radial is anomalous, $\kappa_{ey} = n\chi_e$, and K_q is the collisional energy exchange coefficient.

The ion energy equation is

$$\begin{aligned}\frac{\partial}{\partial t}\left(\frac{3}{2}n_i T_i\right) + \frac{\partial}{\partial x}\left[\frac{5}{2}n_i u_{ix} T_i - \kappa_{jx}\frac{\partial T_i}{\partial x}\right] + \frac{\partial}{\partial y}\left(\frac{5}{2}n_i u_{iy} T_i - \kappa_{jy}\frac{\partial T_i}{\partial y}\right) = \mathbf{u}_i\cdot\nabla p_i \\ + \eta_{ix}\left(\frac{\partial v_{ij||}}{\partial x}\right)^2 + \eta_{iy}\left(\frac{\partial v_{i||}}{\partial y}\right)^2 + K_{qj}(T_e - T_i) + \frac{1}{2}m_i v_{i||}^2 n_i \nu_{iz} + S_{Ej}.\end{aligned} \tag{A7}$$

As for the electrons, the poloidal thermal conduction (and viscosity) coefficients are classical and the radial are anomalous.

The equation for the potential is obtained by subtracting the ion and electron continuity equations, and assuming quasineutrality, $n_i = n_e$:

$$\nabla\cdot\mathbf{J}(\phi) = \frac{\partial}{\partial x}(J_x) + \frac{\partial}{\partial y}(J_y) = 0 \tag{A8}$$

Here by $\mathbf{J}$ we mean the currents excluding the magnetization current since the divergence of the latter is automatically zero owing to it being the curl of a vector. The remaining current components are

$$\mathbf{J} = \left[ne(\mathbf{v}_{i,\nabla B} - \mathbf{v}_{e,\nabla B})\cdot\hat{\mathbf{i}}_x + J_{||}\frac{B_x}{B}\right]\hat{\mathbf{i}}_x + ne(v_{i,y1} - v_{e,y1})\hat{\mathbf{i}}_y. \tag{A9}$$

Note that the terms arising from the $\mathbf{v}_{\nabla B}$-drift do not explicitly depend on ϕ, so they act as source terms in Eq. (A8). The expression for the parallel current, $J_{||}$, comes from the parallel momentum equation for electrons with $m_e \to 0$, yielding,

$$J_{||} = \frac{en}{0.51 m_e \nu_e}\frac{B_x}{B}\left(\frac{1}{n}\frac{\partial P_e}{\partial x} - e\frac{\partial\phi}{\partial x} + 0.71\frac{\partial T_e}{\partial x}\right). \tag{A10}$$

Here ν_e is the electron collision frequency, and the numerical coefficients are described in Ref. 1. Note that the expression for the radial current is different from that in Ref. 4.

The simplest neutral gas model (with nhsp=1 and isupgon=0) considers a diffusive model for the gas transport owing to charge-exchange collisions. In this case, one has the gas velocity given by

$$\mathbf{v}_g = -\frac{\nabla(n_g T_n)}{m_i n_g (n_i \langle \sigma_{cx} v_i \rangle + n_e \langle \sigma_i v_e \rangle)}. \tag{A11}$$

More generally, we use a full momentum equation to describe the parallel neutral velocity together with a diffusive model for the motion perpendicular to the magnetic field; this model is described in Refs. 14–16,18.

The neutral gas density, n_g, is determined by solving the continuity equation

$$\frac{\partial}{\partial t} n_g + \frac{\partial}{\partial x}(n_g v_{nx}) + \frac{\partial}{\partial y}(n_g v_{ny}) = (\langle \sigma_r v_e \rangle - \langle \sigma_i v_e \rangle) n_e n_g. \tag{A12}$$

References

[1] S.I. Braginskii, Transport processes in a plasma *Reviews of Plasma Physics*, Vol. I, Ed. M.A. Leontovich (Consultants Bureau, New York, 1965), p. 205.

[2] T.D. Rognlien and D.D. Ryutov, "Psuedoclassical Transport Equations for Magnetized Edge-Plasmas in the Slab Approximation," Plasma Phys. Reports **25**, 943 (1999).

[3] B.J. Braams, "A Multi-Fluid Code for Simulation of the Edge Plasma in Tokamaks," NET Rept. No. 68, Jan., 1987; "Computational studies in tokamak equilibrium and transport" (thesis, Univ. Utrecht, the Netherlands, 1986).

[4] B.J. Braams, "Radiative Divertor Modeling for ITER and TPX," Contrib. Plasma Phys. **36**, 276 (1996).

[5] P.N. Brown and A.C. Hindmarsh, "Matrix-Free Methods for Stiff Systems of ODEs," SIAM J. Num. Anal. **23**, 610 (1986).

[6] Y. Saad, *Iterative Methods for Sparse Linear Systems* (PWS Pub. Co., Boston, MA, 1996).

[7] T.D. Rognlien, J L. Milovich, M. E. Rensink, and G.D. Porter, "A Fully Implicit, Time Dependent 2-D Fluid Code for Modeling Tokamak Edge Plasmas," J. Nucl. Mater. **196-198**, 347 (1992).

[8] T.D. Rognlien, J.L. Milovich, M.E. Rensink, and T.B. Kaiser, "Simulation of Tokamak Divertor Plasmas Including Cross-Field Drifts," Contr. Plasma Phys. **32**, 485 (1992).

[9] G.D. Porter, M. Fenstremacher, R. Groebner, *et al.*, "Benchmarking UEDGE with DIII-D Data, Contr. Plasma Phys. **34**, 454 (1994).

[10] T.D. Rognlien, P.N. Brown, R.B. Campbell, *et al.*, "2-D Fluid Transport Simulations of Gaseous/Radiative Divertor," Contr. Plasma Phys. **34**, 362 (1994).

[11] G.R. Smith, P.N. Brown, R.B. Campbell, *et al.*,"Techniques and Results of Tokamak-Edge Simulation," J. Nucl. Mat. **220-222**, 1024 (1995).

[12] M.E. Fenstermacher, *et al.*, "UEDGE and DEGAS Modeling of the DIII-D Scrape-Off Layer Plasma," J. Nucl. Mat. **220-222**, 330 (1995).

[13] G.D. Porter, *et al.*, "Simulation of Experimentally Achieved DIII-D Detached Plasmas Using the UEDGE Code," Phys. Plasmas **3**, 1967 (1996).

[14] F. Wising, D.A. Knoll, S. Krasheninnikov, and T.D. Rognlien, "Simulation of Detachment in ITER-Geometry Using the UEDGE Code and a Fluid Neutral Model," Contr. Plasma Phys. **36**, 309 (1996).

[15] F. Wising, D.A. Knoll, S. Krasheninnikov, T.D. Rognlien, and D.J. Sigmar, "Simulation of the Alcator C-Mod Divertor with an Improved Neutral Model," Contr. Plasma Phys. **36**, 136 (1996).

[16] T.D. Rognlien, B.J. Braams, and D.A. Knoll, "Progress in Integrated 2-D Models for Analysis of Scrape-Off Layer Transport Physics," Contr. Plasma Phys. **36**, 105 (1996).

[17] T.D. Rognlien, J.A. Crotinger, G.D. Porter, *et al.*,"Simulation of the Scrape-Off Layer Plasma During a Disruption," J. Nucl. Mat. **241-243**, 590 (1997).

[18] F. Wising, S.I. Krasheninnikov, D.J. Sigmar, D.A. Knoll, T.D. Rognlien, B. LaBombard, B. Lipschultz, and G. McCracken, "Simulation of Plasma Flux Detachment in Alcator C-Mod and ITER," J. Nucl. Mat. **241-243**, 273 (1997).

[19] T.D. Rognlien and D.D. Ryutov, "Analysis of Classical Transport Equations for the Tokamak Edge Plasma," Contr. Plasma Phys. **38**, 152 (1998).

[20] T.D. Rognlien, D.D. Ryutov, and N. Mattor, "Calculation of 2-D Profiles for the Plasma and Electric Field near a Tokamak Separatrix," Czech. J. Phys. **48/S2**, 201 (1998).

[21] M.E. Rensink, L.L. LoDestro, *et al.*, "A Comparison of Neutral Gas Models for Divertor Plasmas," Contr. Plasma Phys. **38**, 325 (1998).

[22] T.D. Rognlien, G.D. Porter, and D.D. Ryutov, "Influence of ExB and Grad-B Terms in 2-D Edge/SOL Transport Simulations," J. Nucl. Mater. **266-269**, 654 (1999).

[23] M.E. Rensink and T.D. Rognlien, "Edge Plasma Modeling of Limiter Surfaces in a Tokamak Divertor Configuration," J. Nucl. Mater. **266-269**, 1180 (1999).

[24] S.I. Krasheninnikov, M.E. Rensink, T.D. Rognlien, *et al.*, "Stability of the Detachment Front in a Tokamak Divertor," J. Nucl. Mater. **266-269**, 251 (1999).

[25] T.D. Rognlien, D.D. Ryutov, N. Mattor, and G.D. Porter, "Two-Dimensional Electric Fields and Drifts near the Magnetic Separatrix in Divertor Tokamaks," Phys. Plasmas **6**, 1851 (1999).

[26] M. Petravic, "Orthogonal Grid Construction for Modeling of Transport in Tokamaks," J. Comp. Phys. **73**, 125 (1987).

[27] G.D. Byrne, "Pragmatic experiments with Krylov methods in the stiff ODE setting," in *Computational Ordinary Differential Equations*, edited by J.R. Cash and I. Gladwell, (Oxford Univ. Press, Oxford, U.K., 1992) p. 323.

[28] P.N. Brown and Y. Saad, "Hybrid Krylov methods for nonlinear systems of equation," SIAM J. Sci. Stat. Comput. **11**, 450 (1990).

[29] A.C. Hindmarsh and A.G. Taylor, "PVODE and KINSOL: Parallel software for differential and nonlinear systems," Tech. Rpt. UCRL-ID-129739, Lawrence Livermore National Lab. (1998).

[30] T.D. Rognlien and X.Q. Xu, "Portable implementation of implicit methods for the UEDGE and BOUT codes on parallel computers," Tech. Rpt. UCRL-ID-133226, Lawrence Livermore National Lab. (1999).

[31] D.P. Stotler, *et al.*, "Coupling of Parallelized DEGAS 2 and UEDGE Codes," Contr. Plasma Phys., to be pub., 2000.

[32] D.P. Stotler, D.E. Post, and D. Reiter, Bull. Am. Phys. Soc. **38** (1993) 1919.

[33] R.A. Hulse, Nucl. Tech./Fusion **3** (1983) 259.

[34] K. Behringer, "Description of the Impurity Transport Code STRAHL," JET Report R(87)08 (1987).